Floating-Point Computations on Reconfigurable Computers

Gerald R. Morris
USACE Army Engineer Research and Development Center, Major Shared Resource Center (ERDC MSRC), Waterways Experiment Station, Vicksburg, MS
gerald.r.morris@erdc.usace.army.mil

Abstract

Modern reconfigurable computers combine general-purpose processors with field programmable gate arrays (FPGAs). The FPGAs are, in effect, reconfigurable application-specific coprocessors. During one run, the FPGA might be a matrix-vector multiply coprocessor; during another run, it might be a linear equation solver. There are several issues associated with the mapping of floating-point computations onto RCs. There is the determination of what the author terms "the FPGA design boundary," i.e., the portion of the application that is mapped onto the FPGA. Furthermore, FPGA-based kernel performance is heavily dependent upon both pipelining and parallelism. The author has coined the phrase "the three p's" to encapsulate this important relationship. In this paper, important FPGA design boundary heuristics are described, and a toroidal architecture and partitioned loop algorithm are used to maximize both pipelining and parallelism for a double-precision floating-point sparse matrix conjugate gradient solver that is mapped onto a reconfigurable computer. Wall clock run time comparisons show that the FPGA-augmented version runs more than two times faster than the software-only version.

1. Background

1.1. Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) were invented in the 1980s by Ross Freeman[10]. As idealized by Figure 1, these semiconductor devices contain configurable logic blocks, fixed logic blocks (multipliers, memories, central processing units, etc.), a programmable interconnection mesh, and programmable input/output (I/O) blocks. The FPGA is programmed via a configuration bitstream to implement complex digital logic circuits. In the traditional FPGA design flow, a hardware description language (HDL) representation of the design is created, and a synthesis tool translates the HDL into netlist files. Netlists, which are essentially text-based descriptions of the schematic, are used by the target-specific place and route (PAR) and bit generation tools to create a configuration bit-stream. At each design stage, the functionality of the design can be verified via simulation. In theory, any digital logic circuit can be placed on an FPGA. In practice, the primary constraints are area, clock rate, and input/output (I/O).

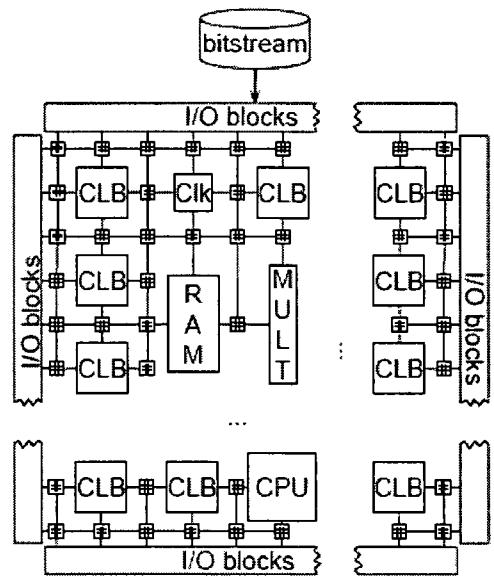


Figure 1. FPGA architecture

1.2. Reconfigurable Computers

The reconfigurable computer (RC) was invented by Gerald Estrin[2] in 1960. An RC is a "fixed plus variable structure" computer that can be "temporarily distorted into a problem oriented special purpose computer." Because of technological limitations, RC research was dormant for over 30 years. However, the FPGA has precipitated a renaissance, and RCs that use general-purpose processors (GPPs) and FPGAs as the fixed plus

0-7695-3088-5/07 $25.00 © 2007 IEEE

variable structure are now available. A typical RC architecture is shown in Figure 2. For applications that have some combination of large-strided or random data reuse, streaming, parallelism, or computationally intensive loops, RCs can achieve higher performance than GPPs. To migrate FPGA-based development out of the hardware design world and into the high-level language (HLL) programming world, there are HLL-to-HDL compilers that provide features such as pipelined loops and parallel code blocks. The goal is to create deeply pipelined, highly parallelized designs without having to deal with the detailed hardware design elements. In concept, designers can develop an algorithm using C, for example, and then compile it into a hardware design. In practice, a hybrid approach involving both HLL and HDL is often necessary.

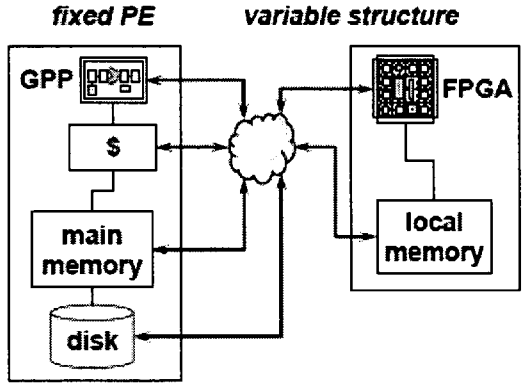


Figure 2. RC architecture

1.3. Sparse Matrix Performance

Applications involving sparse matrices can experience significant performance degradation on cache-based GPPs. The quintessential example is sparse matrix-vector multiply (SMVM), which has a high ratio of memory references to arithmetic operations and suffers from irregular memory access patterns. Over the last 30 years, researchers have tried to mitigate the poor performance of sparse matrix computations through various approaches such as reordering the data to reduce memory bandwidth[1], modifying the algorithm to reuse data[6], and specialized memory controllers[12]. Despite these efforts, sparse matrix performance on GPPs is still dependent upon the sparsity structure of the matrices. In contrast, pipelined FPGA-augmented designs, which have single-cycle memory access, do not depend upon the sparsity structure of the matrix.

1.4. Reduction Problem

Reductions, which occur frequently in scientific computing, are operations such as accumulation that input one or more *n*-vectors and reduce them to a single value. A binary tree of pipelined floating-point cores is a high performance parallel architecture that accepts input vector(s) every clock cycle, and after the pipeline latency, emits one result every clock cycle. To accumulate, say, eight numbers, one can use a binary tree with four adders in the first stage, two adders in the second stage, and a single adder in the third stage. However, because of FPGA area constraints, only small trees will fit on an FPGA. Therefore, designers must translate large parallel reductions into a sequence of smaller reductions and reduce the stream of values that are subsequently produced. Consider the dot product architecture shown in Figure 3. The *n*-vectors, **x** and **y**, are partitioned into *k*-vectors, **u** and **v**. At each clock edge, one pair of *k*-vectors enters the *k*-width dot product unit. When the pipeline is full, the partial dot products, *dj*, stream out, one value per clock cycle. The values in this sequentially delivered vector are accumulated by the adder to produce the dot product, (**x**, **y**). Unfortunately, since the adder is pipelined, the loop introduces a multicycle stage, i.e., a loop-carried dependence. Furthermore, to avoid intermingling, the adder must be flushed after each vector. These stalls result in poor performance and can lead to buffer overruns. Thus, the reduction problem is to reduce multiple sets of sequentially delivered floating-point vectors without stalling the pipeline or imposing unreasonable buffer requirements.

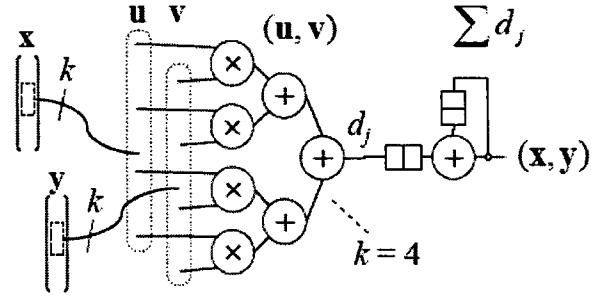


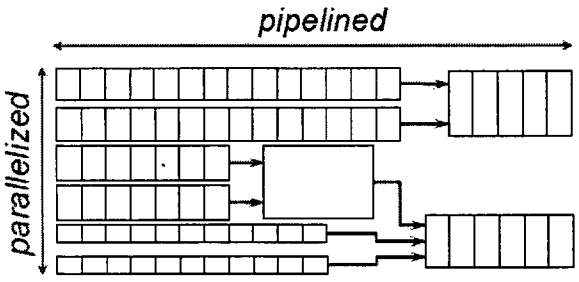
Figure 3. Reduction problem

2. Design Considerations

2.1. Three p's

In order for MHz-rate FPGA-based designs to be competitive with GHz-rate GPP-based designs, the kernels must be both pipelined and parallelized. This idea is conceptually illustrated in Figure 4. The speedup associated with pipelining approaches the pipeline depth (number of pipeline stages) if the pipeline can be kept busy. The speedup associated with parallelization approaches the number of parallel paths. When both pipelining and parallelization are used, the design realizes

a multiplicative speedup that can be symbolized via "the three p's."



Performance ∝ Pipelined×Parallelized

Figure 4. Three p's

2.2. FPGA Design Boundary

The FPGA design boundary deals with determining what elements in an RC-based design are mapped onto the FPGA. As idealized in Figure 5, a designer could choose to map a portion of a module, several modules, or even the complete system onto the FPGA. As with many other engineering decisions, there are no hard and fast rules. Designers must rely on various heuristics. This section describes some of these heuristics.

Obviously, one should consider the anticipated overall speedup. If a given module can be speeded up by a factor of 100 but only constitutes 1 percent of the run time, there does not seem to be much value in mapping it onto an FPGA. In accordance with the three p's mentioned above, designers need to consider the extent to which the module can be both pipelined and parallelized. FPGAs have a limited amount of resources; therefore, the expected size should also be considered. Researchers have shown that control-intensive applications such are sorts do not map well onto FPGAs[4]. Unlike software-based modules, a hardware-based module cannot call other modules, so designers need to either inline the call functionality or make sure the module is a leaf node. Other design considerations include memory bandwidth, the potential for data reuse, the design stability of the algorithm, the efficiency of the algorithm, etc.

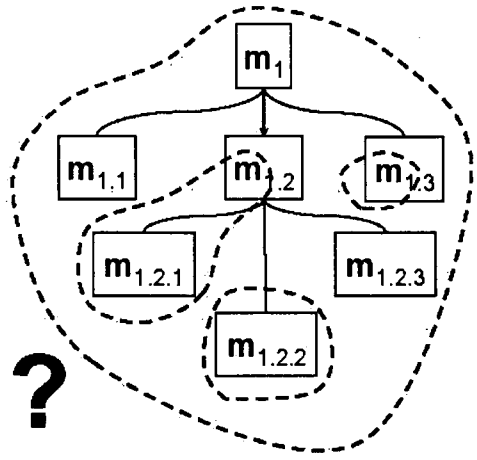


Figure 5. FPGA design boundary

3. Conjugate Gradient Solver

Conjugate gradient (CG), which was discovered in 1952 by Magnus Hestenes and Eduard Stiefel[5], is perhaps the best known iterative method for numerically solving linear equations, $A\mathbf{x} = \mathbf{b}$, whenever A is a symmetric positive-definite (SPD) matrix. A plot of $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}$, where A is an order n SPD matrix, yields an $(n+1)$-dimensional concave-up parabolic surface as depicted in Figure 6. The $\mathbf{x}$ value that minimizes $f(\mathbf{x})$ corresponds to the solution to $A\mathbf{x} = \mathbf{b}$, i.e., the $\mathbf{x}$ value at the lowest point on the surface is the solution. A simplified version of the CG algorithm is shown in Figure 7. The loop calculates the next value of $\mathbf{x}$ (estimated solution), $\mathbf{r}$ (residual), and $\mathbf{p}$ (search direction). Each iteration yields a better $\mathbf{x}$ by "walking downhill" in the A-orthogonal (conjugate) direction given by vector $\mathbf{p}$. A convergence test, as idealized by the while clause at line 5, causes the CG algorithm to terminate.

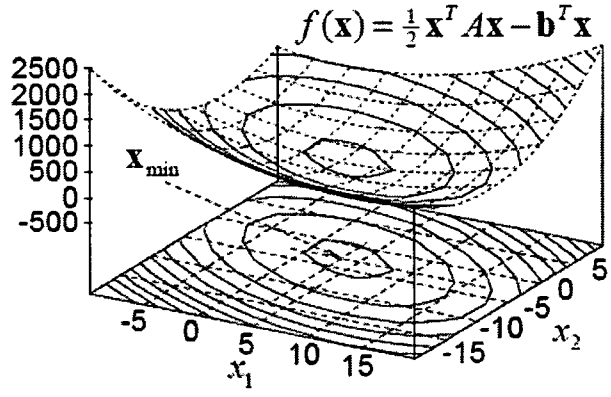


Figure 6. Quadratic form of a vector

```
1:  algorithm CG(A, x, b)
2:      x^(0) ← x_0
3:      p^(0) ← r^(0) ← b ← Ax^(0)
4:      δ ← 0
5:      while (Δ is to big) do
6:          q ← Ap^(δ)
7:          α ← (r^(δ), r^(δ))/(p^(δ), q)
8:          x^(δ+1) ← x^(δ)) + αp^(δ)
9:          r^(δ+1) ← r^(δ)) + αq
10:         β ← (r^(δ+1), r^(δ+1))/ (r^(δ), r^(δ))
11:         p^(δ+1) ← r^(δ+1) + βp^(δ)
12:         δ ← δ + 1
13:     end while
14: end algorithm
```

Figure 7. CG algorithm

3.1. High-Level CG design

A profile of CG[7] shows that it spends over 97 percent of the execution time in Sparse Matrix-Vector Multiply (SMVM) (line 6 of the CG algorithm). Furthermore, SMVM is a relatively small, compute-intensive monolithic code. Finally, for the given application there is a significant potential for data reuse since the A matrix is invariant during all CG iterations. Therefore, SMVM is the module that was targeted for the FPGA. The high-level CG design is shown in Figure 8. The main routine measures how long it takes for CG to solve each set of input equations, $A_i\mathbf{x}_i = \mathbf{b}_i$. A compile-time decision selects the software-only or the FPGA-based version of SMVM. The result vectors, $\mathbf{x}_i$, and performance statistics, Θ_i, are written to output files. Since the A matrix is invariant during the entire CG calculation, the FPGA-based SMVM, which will be described in the next section, pulls a copy of A one time and stores it in local memory for subsequent iterations. Amortization of the matrix transfer cost across all iterations of CG is a key design feature.

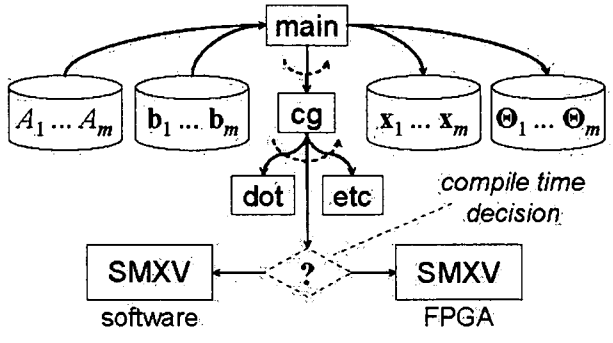


Figure 8. CG design

3.2. FPGA-Based Matrix-Vector Multiply

A block diagram of the FPGA-based SMVM architecture is shown in Figure 9. It consists of a k-width dot product core, an α-stage pipelined adder, a partial reduction buffer (PRB), an output accumulator core, and some on-chip and local memory banks. The input sparse matrix, A, is represented in compressed sparse row (CSR) format via the three vectors:

- **val** – the row-wise matrix values;
- **col** – the column index of each value; and
- **ptr** – the position in the **val** vector where each row begins.

The basic algorithm for each row is to calculate a series of partial dot products, d_{ih}, and reduce them to the single value, $y_i = \sum_h d_{ih} = \sum_j a_{ij}x_j$. The k-width dot product unit accepts two double-precision floating-point k-vectors every clock cycle. The **u** inputs from **val** correspond to the next k elements of the A matrix. The corresponding k values from **col** ensure that the matching k elements of **x** are sent to the **v** inputs. After the latency, a sequential stream of partial dot products are emitted.

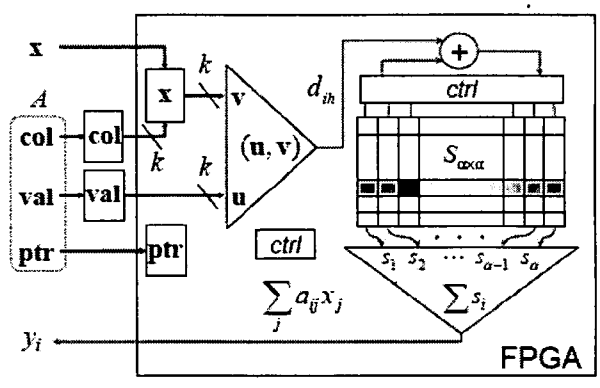


Figure 9. SMVM processor

To reduce the partial dot products, there is an α-stage pipelined adder and a constant-sized α-row by α-column PRB. A round-robin scheduling algorithm guarantees an α-cycle interval between subsequent references to the same memory location in S. The binary tree output accumulator reduces completed rows of S to produce the components of vector **y**. The easiest way to envision the round-robin partial summation algorithm is to view the toroidal access pattern of the PRB shown in Figure 10. The accumulation of a given input vector is restricted to a specific row, e.g., the gray row, within the PRB. Even if there are more than α elements in the input vector, the major circumference of the torus (number of columns) is α, thereby ensuring that any previous data at a given location, e.g., the black square, have already traversed the adder and been written back by the time that location is

again referenced. If a series of small vectors are to be reduced, the minor circumference of the torus (number of rows) ensures that by the time a row needs to be reused, its contents have already been sent to the output accumulator and the row initialized to zero. This toroidal access pattern makes S appear to be an infinite two-dimensional array, which can handle arbitrary sets of sequentially delivered vectors without stalling the pipeline.

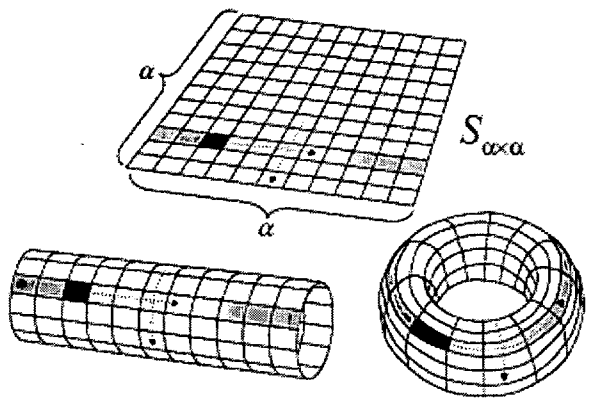


Figure 10. Toroidal access pattern of PRB

4. Implementation and Results

4.1. Target RC and Implementation

An SRC-6 MAPStation[9] was used as the target RC. It has dual 2.8GHz Xeon GPPs with 512KB cache and 1GB RAM. The MAP Series MPC processor contains two Xilinx Virtex II 6,000 FPGAs running at 100MHz. Each FPGA has 288KB of on-chip BRAM, and six banks of local memory provide an additional 24MB of memory. The SRC Carte C compiler v2.1 and Xilinx ISE v7.2 were used for the FPGA modules, and the Intel C compiler v8.1 was used for the software modules. The dot product unit and output accumulator were implemented using VHDL and the IEEE-Std-754 double-precision floating-point cores described in Reference 3. They were synthesized using Synplify Pro v8.1 and integrated into the Carte environment as user-defined macros.

Since CG is difficult to implement properly[8], an off-the-shelf implementation from the SPARSKIT[11] library was used as a baseline. The optimized software SMVM that came with SPARSKIT was also used. The target RC allowed for matrices up to order $n = 4096$. The limiting factor was the number of simultaneous local memory reads. It was necessary to store some vectors, e.g., **x**, in the FPGA block memories. Future RCs will likely have a larger number of local memory banks that can handle significantly larger problems.

4.2. Experimental Results

For each matrix order, 1,000, 2,000, 3,000, and 4,000, three SPD matrices that have sparsity percent values of two, four, and six percent were generated. For example, the two percent sparsity test matrix for the n = 1,000 case contains $n_z = n^2 \times 2\% = 10^6 \times 0.02 = 20K$ nonzero entries. The resulting twelve SPD matrices were used as inputs for the two versions of CG. To capture the entire system behavior including data transfer time to and from the FPGA-based modules, the main routine was instrumented with microsecond-resolution timers to capture the wall clock run time of the entire application. Figure 11 compares the wall clock run time of the FPGA-augmented version with the software-only version. For the 1K(*) cases, which fit in the 512KB cache of the Xeon, the software-only version of CG has the best performance. However, for the remaining test cases, the FPGA-augmented version of CG outperforms software.

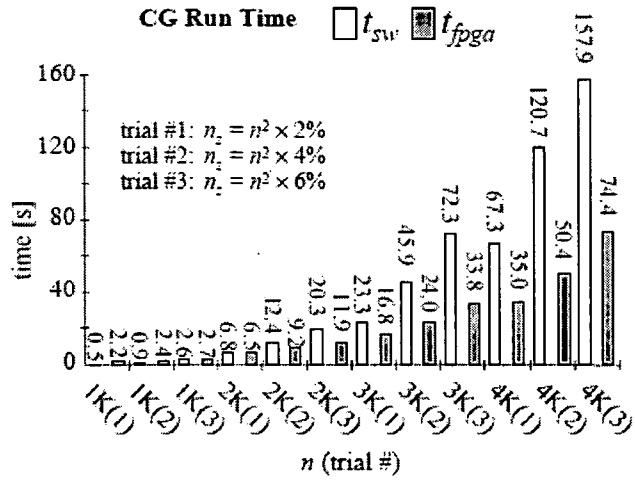


Figure 11. Run time comparison

5. Future Work

A recurring limitation is the number of local memory banks needed to provide the parallelism associated with high performance FPGA kernels. Next-generation RCs will have a significantly larger number of memory banks. The soon-to-be-released SRC-7, for example, supports 20 simultaneous memory reads, as opposed to the 6 simultaneous reads in the SRC-6. In addition, because of the deeply pipelined floating-point cores used on FPGAs, it is unlikely that the 100-fold speedups that have been demonstrated for integer applications can be achieved for floating-point applications. However, 10-fold overall speedups may be possible. The most obvious future work is to reconsider the current designs by moving the on-chip stores into the local memory banks and to increase the

data path width (parallelism). These two considerations should result in significant speedups and accommodate much larger matrices.

6. Conclusions

In this paper, important FPGA design boundary heuristics are described, and a toroidal architecture and partitioned loop algorithm are used to maximize both pipelining and parallelism for a double-precision floating-point sparse matrix conjugate gradient solver that is mapped onto a reconfigurable computer. Wall clock run time comparisons show that the FPGA-augmented version runs more than two times faster than the software-only version.

Despite the limitations in the current generation RCs, this work and related research efforts provide strong evidence that FPGA-augmented RCs may be the next wave in the quest for higher floating-point performance.

References

1. Cuthill, E. and J. McKee, "Reducing the bandwidth of sparse symmetric matrices." In *Proceedings of the 1969 24th National Conference of the ACM*, San Francisco, CA, USA, August 1969.

2. Estrin, G., "Organization of computer systems–the fixed plus variable structure computer." In *Proceedings of the Western Joint Computer Conference*, San Francisco, CA, USA, May 1960.

3. Govindu, G., R. Scrofano, and V.K. Prasanna, "A library of parameterizable floating-point cores for FPGAs and their application to scientific computing." *In Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms*, Las Vegas, NV, USA, June 2005.

4. Harkins, J., T. El-Ghazawi, E. El-Araby, and M. Huang, "Performance of sorting algorithms on the SRC 6 reconfigurable computer." In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology (FPT'05)* Singapore, pp. 295–296,, December 2005.

5. Hestenes, M. and E. Stiefel, "Methods of conjugate gradients for solving linear systems." *Journal of Research of the National Bureau of Standards*, 49(6), December 1952.

6. Im, E.J., K.A. Yelick, and R. Vuduc, "SPARSITY: An optimization framework for sparse matrix kernels." *International Journal of High Performance Computing Applications*, 18(1), February 2004.

7. Morris, G.R., R.D. Anderson, and V.K. Prasanna, "A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer." In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, USA, April 2006.

8. O'Leary, D. M, "Methods of conjugate gradients for solving linear systems." In D.R. Lide, editor, *A Century of Excellence in Measurements, Standards, and Technology A Chronicle of Selected NBS/NIST Publications, 1901–2000*, NIST Special Publication 958, 2001.

9. SRC Computers, Inc., *General purpose reconfigurable computing systems*, http://www.srccomp.com.

10. Xilinx, Inc., *How Xilinx began. In Xilinx: our history*, http://www.xilinx.com/company/history.htm, 2006.

11. Saad, Y., "SPARSKIT: a basic tool kit for sparse matrix computations." http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html, June 1994.

12. Zhang, L., Z. Fang, M. Parker, B.K. Mathew, L. Schaelicke, J.B. Carter, W.C. Hsieh, and S.A. McKee, "The impulse memory controller." *IEEE Transactions on Computers*, 50(11), November 2001.